Exo 1 :

```python
print( "Entrer un entier naturel" )
n = int( input() )
som = 0
for i in range( 1, n ) :
    som = som + i
print( f"La somme vaut {som}" )
```

Exo 2 :

a) Déroulé ci-contre.

b) Ce code permet d'afficher les diviseurs strictement inférieurs et positifs de n.

c) Oui, puisque les diviseurs de n distincts sont nécessairement inférieurs à $\frac{n}{2}$. Il suffit décrire `d <= n//2`.

ligne	n	d	d < n	n%d==0	sortie
1	6				
2	-				1
3	-	2			-
4	-	-	True		-
5	-	-	-	True	-
6	-	-	-	-	2
7	-	3	-	-	-
4	-	-	True	-	-
5	-	-	-	True	-
6	-	-	-	-	3
7	-	4	-	-	-
4	-	-	True	-	-
5	-	-	-	False	-
7	-	5	-	-	-
4	-	-	True	-	-
5	-	-	-	False	-
7	-	6	-	-	-
4	-	-	False	-	-

Exo 3 :

```python
print( f(y) )          # affiche 7
print( g([3,4,5,6]) ) # affiche 2
print( h(1) )          # affiche 3
print( x, y, z )       # affiche 0 1 2
print( tab )           # affiche [1,1,5,7]
```

Exo 4 : Rq : un code n'est pas nécessairement une fonction.

a)

```python
u = 4
for i in range(20):
    u = u**2 -3
print( u )
```

b) On peut proposer une de ces deux solutions

```python
u = 4
for k in range(21):
    if k%2==0:   # k est d'indice pair
        print( u )
    u = u**2 -3
```

```python
u = 4
print( u )      # u_0
for k in range(1,21,2): # k in range(10)
    u = u**2 -3  # u d'indice impair
    u = u**2 -3  # u d'indice pair
    print( u )   # k est d'indice pair
```

c) On utilise une boucle `while`

```python
u, k = 4, 0
while u<100:
    u = u**2 -3
    k += 1
print( k )
```

Exo 5 : On peut proposer deux solutions

```python
def suiteU(n: int) -> list:
    un, un1 = 1, 1   # u_n, u_(n+1)
    if n==1: return [ un ]
    tab = [un, un1]
    for k in range(2, n):
        tmp = un1
        un1 = un1 - un + 2
        un = tmp
        tab += [ un1 ]
    return tab
```

```python
def suiteU(n: int) -> list:
    tab = [1]*n  # le cas n==1 est traité !
    for i in range(2,n):
        tab[i] = tab[i-1] - tab[i-2] + 2
    return tab
```

Exo 6 :

a)

```python
def nbOcc(let: str, phrase: str) -> int:
    nocc = 0
    for l in phrase:
        if let==l:
            nocc += 1
    return nocc
```

b) On peut proposer un code qui ne fonctionne que pour un `mot` de deux lettres (les deux premiers codes), ou un code `nbOccG` qui généralise le problème à un `mot` d'un nombre quelconque de lettres.

```python
def nbOcc2(mot: str, phrase: str) -> int:
    nocc = 0
    for i in range( len(phrase)-1 ):
        if mot[0]==phrase[i] and mot[1]==phrase[i+1]:
            nocc += 1
    return nocc
```

```python
def nbOcc2(mot: str, phrase: str)-> int:
    nocc = 0
    for i in range( len(phrase)-1 ):
        if mot==phrase[i]+phrase[i+1]:
            nocc += 1
    return nocc
```

```python
def nbOccG(mot: str, phrase: str)-> int:
    n = len(mot)
    nocc = 0
    for i in range( len(phrase)-n+1 ):
        j = 0
        while j<n and phrase[i+j]==mot[j]:
            j += 1
        if j == n:
            nocc += 1
    return nocc
```

Exo 7 :

a) On a $(c_1,t_1) = (65, 254)$ (était également accepté $(65, 256)$)

b)

```python
def crocdoux(n: int) -> tuple:
    c = 50
    t = 200
    for i in range( n ) :
        tmp = c // 2 + t // 5
        t = 3 * t // 2 + 20 - 4 * c // 3
        c = tmp
    return c, t
```

Exo 8 : (***BONUS***) Le carré est assez simple, on propose

```python
n = int(input())
print( n*"*" )
for i in range(n-2) :
    print( "*" + (n-2)*" " + "*" )
print( n*"*" )
```

L'étoile est plus délicate puisqu'il ne faut avoir qu'une seul étoile au centre. On pouvait proposer

```python
n = int(input())
for k in range( n//2 ) :
    print( k*" " + "*" + (n-2-2*k) * " " + "*")
print( (n//2)*" " + "*")
for k in range( n//2-1, -1, -1 ) :
    print( k*" " + "*" + (n-2-2*k) * " " + "*")
```